Guarded Type Promotion

Eliminating Redundant Casts in Java

Johnni Winther
Department of Computer Science
Aarhus University
jw@cs.au.dk

ABSTRACT

In Java, explicit casts are ubiquitous since they bridge the gap between compile-time and runtime type safety. Since casts potentially throw a ClassCastException, many programmers use a defensive programming style of *guarded casts*. In this programming style casts are protected by a preceding conditional using the instanceof operator and thus the cast type is redundantly mentioned twice. We propose a new typing rule for Java called Guarded Type Promotion aimed at eliminating the need for the explicit casts when guarded. This new typing rule is backward compatible and has been fully implemented in a Java 6 compiler. Through our extensive testing of real-life code we show that guarded casts account for approximately one fourth of all casts and that Guarded Type Promotion can eliminate the need for 95 percent of these guarded casts.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Java, type cast, type checking

1. INTRODUCTION

When teaching the Java instanceof expression one might, to illustrate a common use case, present an example like

```
if (o instanceof Foo) {
  ((Foo)o).doFooStuff();
}
```

and be met with the response: "Why do we *need* to say that o is of type Foo when we have just gotten a 'yes' from asking the *exact same question*?" And yes, why do we?

FTfJP'11, July 26, 2011, Lancaster, UK.

The example is an instance of a programming style we call *guarded casts* in which a cast is preceded by a conditional to ensure the cast validity. Since such casts contain redundant type information they are not only tedious to write but also subject to errors.

We present a new backwards compatible typing for Java aimed at eliminating the need for these redundant casts. Our approach is to extend Java with a path sensitive typing of local variables and final fields called Guarded Type Promotion. We use the type information provided by the runtime checks to promote the types of such expressions thus enabling a more precise typing without the need for further annotations.

Guarded Type Promotion only targets casts that have been *explicitly* guarded, like in the previous example. This is because we do *not* want to support casts in general but only those part of this defensive programming style. For guarded casts, however, Guarded Type Promotion not only eliminates the need for the redundancy but also ensures the cast validity statically.

Guarded Type Promotion is similar to type inference as for instance proposed by Peter von der Ahé [3]. One could argue that it is just a type inference that did not go all way; that the rules should and could just as well handle code like

```
Object o = "foo";
o.trim(); // o promoted to String by "foo"
```

Guarded Type Promotion is not just an inferior type inference, it is *intentionally* not using all type knowledge present within the code. The main difference between type inference and Guarded Type Promotion is that the programmer through a guard expresses the intent to couple the code with a more specific type than the static type implies. The use of generalized types is a key component to decoupling and the inference of a more precise type for all expressions would counter this meta-pattern. For instance, an important reason for choosing a typing like

```
List<String> list = new LinkedList<String>();
```

over

```
LinkedList<String> list = new LinkedList<String>();
```

is the exchangability of implementation that the generalization enables and Guarded Type Promotion supports this.

The rest of this paper is structured as follows: In Section 2 we give an overview of casts in Java. In Section 3 we present the data-flow analysis behind the promotion. Some of the transfer functions are shown in Section 4 and in Sec-

tion 5 we show how the promotion is integrated into Java. Experiments are presented in Section 6 and in Section 7 we discuss related and future work.

2. CASTS

To get an overview of how casts occur in Java we categorize the explicit casts into four groups according to their runtime type safety.

2.1 Guarded Casts

The first group of casts is the *guarded casts* where the programmer performs a runtime check before casting and thus ensures the validity of the cast by guarding it. The guarding of a cast can be done using the instanceof expression or by using the .class constants and the Class⟨?⟩ values returned by Object.getClass(). The use of the guarded cast programming idiom falls into 4 categories that are all targeted by Guarded Type Promotion.

An *if-guarded cast* is a cast which is preceded by a runtime check of the target against the same type. For instance as in the code below where the cast (Foo)o is guarded by the preceding o instanceof Foo expression, possibly using the ternary operator ? : or lazy operators.

```
if (o instanceof Foo) {
  Foo foo = (Foo)o;
  ...
}
```

```
if (o instanceof Foo &&
      ((Foo)o).isBar()) {
  ...
}
```

```
Bar bar = o instanceof Foo ? ((Foo)o).getBar() : null;
```

A *dead-if-guarded cast* is a cast where the guard uses reachability to ensure the validity. For instance

```
if (!(o instanceof Foo)) {
  return;
}
Foo foo = (Foo)o;
...
```

where the cast is unreachable if o is *not* a Foo instance.

An *ensure-guarded cast* is a cast in which both paths through a conditional result in the target having the intended type. For instance

```
if (!(o instanceof Foo)) {
  o = new Foo();
}
Foo foo = (Foo)o;
...
```

where o is intended to have type Foo and where the if statement ensures just this.

A *while-guarded cast* is a cast in which a loop ensures the intended type. For instance

```
while (o != null && !(o instanceof Foo)) {
  o = o.parent();
}
Foo foo = (Foo)o;
...
```

where the parent() relation of o is traversed until a Foo instance or null is found.

2.2 Semi-Guarded Casts

Semi guarded casts are guarded by the semantics of the program but the cast validity is invisible at the type level. For instance

```
Foo foo = ...
if (foo.isBar()) {
  Bar bar = (Bar)foo;
  ...
}
```

where the call to isBar() ensures that foo is an instance of Bar. Since semi-guarded casts are invisible at the type level these casts are not targeted by our type promotion.

2.3 Unguarded Casts

Unguarded casts are casts whose validity is not ensured locally. These come in many flavors. For instance unchecked casts or casts resulting from the use of raw types, like in

```
List list = ... { // a list of Foo elements
for (Object o : list) {
  Foo foo = (Foo)o;
  ...
}
```

where each retrieved element of the list of Foo elements need to be cast before use.

The validity of many unguarded casts is ensured by invariants that are known to the programmer but not ensured locally. A classic example of this is the casting of a call to clone() to the type of the receiver, for instance

```
Calendar copy = (Calendar)calendar.clone();
```

where the return type is known to be of type Calendar despite its signature, Object clone().

A notable kind of unguarded casts are *invalid casts* which are guarded casts gone bad. Like in the example below, this occurs when the instanceof type and the cast type differ. Invalid casts are thus caused by the redundancy of guarded casts.

```
Object o = ...
if (o instanceof String) {
  // a String instance can not be an Integer instance
  Integer i = (Integer)o;
  ...
}
```

During our experiments, our type promotion revealed 4 such invalid casts.

2.4 Safe Casts

The last group is the *safe casts* which are casts that cannot cause an exception at runtime. The only safe casts are *upcasts* which are casts to super types. When such a cast is not needed for instance for disambiguating overloaded methods, an upcast is also often referred to as an *unnecessary cast*.

For completeness we mention *primitive conversions* like (char)42 or *boxing conversions* like (Integer)42 which despite their syntactical likeness are not casts but conversions. For this reason these are not included in the figures presented in Section 6.

3. TYPE PROMOTION

Guarded Type Promotion is a path sensitive typing. This means that it contains an intraprocedural data-flow analysis which collects type information for certain expressions. This information enables the computation of a *promoted type* which is more specific than the static type of the expression while ensuring that the dynamic type is always a subtype of the promoted type. We refer to the static type as the *declared type* since the promoted type is also statically known.

$$\mathit{targetOf}(\mathsf{v}) = \mathit{Local}(\mathsf{v})$$

$$\frac{\mathit{targetOf}(\mathsf{e}) = t}{\mathit{targetOf}((\mathsf{e})) = t} \qquad \frac{\mathit{targetOf}(\mathsf{e}) = t}{\mathit{targetOf}((\mathsf{T})\mathsf{e}) = t}$$

$$\frac{\mathit{targetOf}(\mathsf{e}_1) = t}{\mathit{targetOf}(\mathsf{e}_1 = \mathsf{e}_2) = t}$$

$$\frac{\mathit{targetOf}(\mathsf{e}) = t \qquad \mathsf{f}\ \text{final}}{\mathit{targetOf}(\mathsf{e.f}) = \mathit{Field}(t, \mathsf{f})}$$

$$\frac{\mathsf{f}\ \text{final}}{\mathit{targetOf}(\mathsf{T.f}) = \mathit{StaticField}(\mathsf{T}, \mathsf{f})}$$

Figure 1: Target Rules

Targets.

If the guarded expression is a local variable access that has not been updated since the guard or a final field access then it is sound to use the promoted type. If the expression is a non-final field access, an array access, or a method invocation then it is *unsound* to use the promoted type. A non-final field or an array might be updated non-locally by a method invocation or even a different thread and a method invocation might return different values on each invocation.

To increase the syntactical flexibility of the analysis, expressions like for instance o, (o), and $\mathsf{o} = \mathsf{e}$ must be regarded as the same target; a test of any of these is a test of the value of o. The promotable expressions are therefore mapped into a *target* of the following grammar by the rules in Figure 1.

$$\begin{array}{rcll} t & ::= & \mathit{Local}(\mathsf{v}) & \text{local variable } \mathsf{v} \\ & | & \mathit{Field}(t, \mathsf{f}) & \text{field } \mathsf{f} \text{ on target } t \\ & | & \mathit{StaticField}(\mathsf{T}, \mathsf{f}) & \text{static field } \mathsf{f} \text{ on type } \mathsf{T} \end{array}$$

The Promotion Analysis.

The type promotion analysis is inspired by the reachability and definite assignment analyses defined in §14.21 and §16 of [5]. For each statement and non-boolean expression the analysis computes two *type maps* $\mathcal{B}[\![\cdot]\!]$ and $\mathcal{A}[\![\cdot]\!]$ containing the promoted type information *before* and *after* the statement/expression, respectively, and for each boolean expression it computes three type maps $\mathcal{B}[\![\cdot]\!]$, $\mathcal{A}_t[\![\cdot]\!]$ and $\mathcal{A}_f[\![\cdot]\!]$ containing the promoted type information *before* the expression, after the expression *when true* and after the expression *when false*, respectively.

The runtime type information available in Java does *not* include the type arguments but merely the type declaration of which an object is an instance. Therefore, a guard, regardless of whether we are using instanceof, getClass(), or .class, only provides declarational information about the runtime type of a value. A declaration d of a type τ, denoted $\mathit{declOf}(\tau)$ is thus similar to raw types with for instance $\mathit{declOf}(\mathsf{List}\langle\mathsf{String}\rangle) = \mathsf{List}$ and two declarations, d_1 and d_2, have the natural ordering $d_1 \lhd d_2$ if d_1 inherits from d_2. We define the type of a declaration d, denoted $\mathit{typeOf}(d)$, as the instantiation of d in which all type arguments are unbounded wildcards. For instance $\mathit{typeOf}(\mathsf{List}) = \mathsf{List}\langle?\rangle$.

A type map, m, is a map from targets to *target info* constructs.[1] A target info construct $i = \mathit{targetInfo}(\tau_d, H, D)$ holds 3 parts of information. First the declared type, τ_d, of the target for convenience. Secondly a set of *type holder* constructs, H, which contain type information for disjoint paths. For instance that a target is either a String instance, an Integer instance or null. Thirdly a set of *declarations of interest*, D, which is the declarations that the target has been tested against.

A type holder $h = \mathit{typeHolder}(\tau_d, D_t, D_f)$ contains type information for one possible path. The declared type, τ_d, is used to calculate the promoted type of the type holder. The set D_t is called *the set of true declarations* and is the set of declarations that the target is known to be an instance of, i.e. a non-null subtype of, on this path. The set D_f is called *the set of false declarations* and is the set of declarations that the target is known *not* to be an instance of. For instance a type holder $\mathit{typeHolder}(\mathsf{Object}, \{\mathsf{Number}\}, \{\mathsf{Integer}\})$ tells us that the target is known to be an instance of Number but *not* an instance of Integer.

The Promoted Type.

The promoted type of a target in the type map mapped to $\mathit{targetInfo}(\tau_d, H, D)$ is computed as the least upper bound, *lub*, of the types of its type holders:

$$\mathit{typeOf}(\mathit{targetInfo}(\tau_d, H, D)) = \mathit{lub}(\{\mathit{typeOf}(h) | h \in H\})$$

The calculation of the type of a type holder is quite complicated. For brevity it is shown here in a simplified form:

```
typeOf(typeHolder(τ_d, D_t, D_f)):
  if ∃d ∈ D_f. declOf(τ_d) ◁ d
    return ⊥
  return glb({τ_d} ∪ {typeOf(d) | d ∈ D_t})
```

If the target is known *not* to be of its declared type then the type is the null-type, $\bot$. Otherwise the *glb* function computes the greatest lower bound of the declared type and the types of the true declarations. The promoted type is thus always a subtype of the declared type.

The actual *typeOf* function on type holders can also infer type arguments from the declarations. Consider for instance the following code:

```
interface A⟨X⟩ {}
interface B extends A⟨B⟩ {}
interface C⟨X⟩ extends A⟨X⟩ {}

Object o = ...
if (o instanceof C) { // o has promoted type C⟨?⟩
  if (o instanceof B) { // o has promoted type C⟨B⟩ & B
```

Here the type argument of C is gradually inferred by the information provided by C and B.

The Join Function.

The *join* function computes the least upper bound of two type maps. It is defined as

```
join(m_1, m_2):
  return {t ↦ join(i_1, i_2) | t ↦ i_1 ∈ m_1, t ↦ i_2 ∈ m_2}
```

where the least upper bound of two target info constructs is

```
join(targetInfo(τ_d, H_1, D_1), typeHolder(τ_d, H_2, D_2)):
  return targetInfo(τ_d, H_1 ∪ H_2, D_1 ∪ D_2)
```

The *join* function is used to merge two paths in the dataflow analysis and for a boolean expression e we thus define its after map, for when we are *not* taking its value into account,

[1] We write $m \setminus \{t \mapsto i\}$ for the map m in which the mapping for t is replaced by $t \mapsto i$ and $m \setminus \{t \mapsto \bullet\}$ for the map m in which the mapping for t is removed.

as $\mathcal{A}[\![\mathsf{e}]\!] = \mathit{join}(\mathcal{A}_t[\![\mathsf{e}]\!], \mathcal{A}_f[\![\mathsf{e}]\!])$.

The Promote Function.

New type information for a target is stored in the type map through the *promote* function which is defined as follows

promote(m,t,D_t,D_f,τ_d):
  if $t \mapsto i \in m$ then // promote existing target
    return $m \setminus \{t \mapsto \mathit{promote}(i, D_t, D_f)\}$
  else // add new promotion
    $h \leftarrow \mathit{typeHolder}(\tau_d, D_t, D_f)$
    return $m \cup \{t \mapsto \mathit{targetInfo}(\tau_d, \{h\}, D_t \cup D_f)\}$

with the promotion of a target info construct defined as

promote(*targetInfo*(τ_d, H, D),D'_t,D'_f):
  for $\mathit{typeHolder}(\tau_d, D_t, D_f) \in H$
    $h_i \leftarrow \mathit{typeHolder}(\tau_d, D_t \cup D'_t, D_f \cup D'_f)$
  $H' \leftarrow \{h_i \mid h_i$ is not invalid$\}$
  if $|H'| > 0$ // combine valid paths
    return $\mathit{targetInfo}(\tau_d, H', D \cup D'_t \cup D'_f)$
  else // create assumed path
    // mark as erroneous
    $h \leftarrow \mathit{typeHolder}(\tau_d, D'_t, D'_f)$
    return $\mathit{targetInfo}(\tau_d, \{h\}, D \cup D'_t \cup D'_f)$

The promotion of a target info attempts to combine the new type information with that of all of its possible paths, keeping only those paths that are not invalid. A type holder is invalid if it contradicts itself, that is, if it for instance requires the target to be both Integer and not Object, i.e. both null and non-null at the same time, or for instance requires the target to be both Integer and String at the same time.

If no paths are valid, a target info construct assuming the new type information is created. Since it represents an invalid state it is marked as erroneous. If a type map contains erroneous target info after the data-flow analysis has reached a fixed-point, an 'unreachable code' error is produced. During our experiments one such error was flagged, revealing a sequence of contradicting guards.

4. TRANSFER FUNCTIONS

In this section we present the transfer functions for an interesting subset of the Java constructs. Code examples involving most of these constructs are shown in Appendix A. For brevity we denote the type maps of the construct in question as $\mathcal{A}[\![\cdot]\!]$, $\mathcal{B}[\![\cdot]\!]$, etc.

• e instanceof τ

The instanceof expression is the primary source of promotional information. For this we define the transfer function as follows:

$$\begin{array}{rcl} \mathcal{B}[\![\mathsf{e}]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{A}_t[\![\cdot]\!] & = & \mathit{promote}(\mathcal{A}[\![\mathsf{e}]\!], \mathit{targetOf}(\mathsf{e}), \{\mathit{declOf}(\tau)\}, \emptyset) \\ \mathcal{A}_f[\![\cdot]\!] & = & \mathit{promote}(\mathcal{A}[\![\mathsf{e}]\!], \mathit{targetOf}(\mathsf{e}), \emptyset, \{\mathit{declOf}(\tau)\}) \end{array}$$

If the instanceof expression results in true at runtime, we know that e *is* an instance of the declaration of τ and we therefore promote the target of e with the set $\{\mathit{declOf}(\tau)\}$ of *true* declarations. Otherwise, if the instanceof expression results in false at runtime, we know that e *is not* an instance of $\mathit{declOf}(\tau)$ and we therefore promote the target of e with the set $\{\mathit{declOf}(\tau)\}$ of *false* declarations.

• e_1.getClass() == e_2.getClass()

Another kind of guard is an expression where two calls to the special Object.getClass() method are tested for equality.

$$\begin{array}{rcl} \mathcal{B}[\![\mathsf{e}_1]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{B}[\![\mathsf{e}_2]\!] & = & \mathcal{A}[\![\mathsf{e}_1]\!] \\ m & = & \mathit{promote}(\mathcal{A}[\![\mathsf{e}_2]\!], \mathit{targetOf}(\mathsf{e}_1), \{\mathit{declOf}(\mathsf{e}_2)\}, \emptyset) \\ \mathcal{A}_t[\![\cdot]\!] & = & \mathit{promote}(m, \mathit{targetOf}(\mathsf{e}_2), \{\mathit{declOf}(\mathsf{e}_1)\}, \emptyset) \\ \mathcal{A}_f[\![\cdot]\!] & = & \mathcal{A}[\![\mathsf{e}_2]\!] \end{array}$$

If the result of such an expression is true, we know that the two receiver expressions, e_1 and e_2, are instances of the same declaration. In the $\mathcal{A}_t[\![\cdot]\!]$ map the true set of the target of e_1 is therefore promoted with the declaration of e_2, $\mathit{declOf}(\mathsf{e}_2)$, and vice versa. In the case of false we have no further knowledge of the type of e_1 and e_2 since we only know that they are instances of different *but unknown* types. Tests against T.class expressions have a similar promoting transfer function.

• e == null

Test for nullness also promotes a target. This is necessary to eliminate while-guarded casts. If e is null then its false declarations are extended with $\{\mathit{declOf}(\mathsf{e})\}$, i.e. e is *not* an instance of its declared type. If e is *not* null we instead extend the true declarations with $\{\mathit{declOf}(\mathsf{e})\}$.

$$\begin{array}{rcl} \mathcal{B}[\![\mathsf{e}]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{A}_t[\![\cdot]\!] & = & \mathit{promote}(\mathcal{A}[\![\mathsf{e}]\!], \mathit{targetOf}(\mathsf{e}), \emptyset, \{\mathit{declOf}(\mathsf{e})\}) \\ \mathcal{A}_f[\![\cdot]\!] & = & \mathit{promote}(\mathcal{A}[\![\mathsf{e}]\!], \mathit{targetOf}(\mathsf{e}), \{\mathit{declOf}(\mathsf{e})\}, \emptyset) \end{array}$$

• e_1 && e_2

Similarly to the definite assignment analysis in Java the transfer functions for &&, ||, and the conditional operator, ? :, take the conditional flow into account. For instance the transfer function for && is defined as follows:

$$\begin{array}{rcl} \mathcal{B}[\![\mathsf{e}_1]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{B}[\![\mathsf{e}_2]\!] & = & \mathcal{A}_t[\![\mathsf{e}_1]\!] \\ \mathcal{A}_t[\![\cdot]\!] & = & \mathcal{A}_t[\![\mathsf{e}_2]\!] \\ \mathcal{A}_f[\![\cdot]\!] & = & \mathit{join}(\mathcal{A}_f[\![\mathsf{e}_1]\!], \mathcal{A}_f[\![\mathsf{e}_2]\!]) \end{array}$$

• $\mathsf{e}_1 = \mathsf{e}_2$

To handle assignment soundly we must take into account the side effects of the update of the target t of the left-hand side expressions e_1. Since for instance e_1.f potentially has a different value if the value of e_1 is changed, we remove from the type map all targets of which t is a proper subtarget.

$$\begin{array}{rcl} \mathcal{B}[\![\mathsf{e}_1]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{B}[\![\mathsf{e}_2]\!] & = & \mathcal{A}[\![\mathsf{e}_1]\!] \\ t & = & \mathit{targetOf}(\mathsf{e}_1) \\ m & = & \mathcal{A}[\![\mathsf{e}_2]\!] \setminus \{t' \mapsto \bullet \mid t' \text{ is a proper subtarget of } t\} \\ \mathcal{A}[\![\cdot]\!] & = & \mathit{reduce}(m, t, \mathit{typeOf}(\mathcal{A}[\![\mathsf{e}_2]\!])) \end{array}$$

For the target t itself we could soundly just discard the promotion but we can do something better. Through the *reduce* function defined below we set the promoted information to be based on the most specific declaration between the right-hand side type and declarations of interest found in the target info.

```
reduce(m, t, d):
  if t ↦ targetInfo(τ_d, H, D) ∈ m then
    D' ← {d' ∈ D | d ◁ d'}
    if |D'| > 0 // is the declaration of interest?
      h_t ← typeHolder(τ_d, D', ∅) // t is an instance of d
      h_f ← typeHolder(τ_d, ∅, D') // t is null
      i ← targetInfo(τ_d, {h_t, h_f}, D)
      return m \ {t ↦ i}
    else // discard
      return m \ {t ↦ •}
  else
    return m
```

For instance by using the combined knowledge that we are interested in whether o is of type Foo and that the right-hand side is an instance of Foo or null, we can keep the partial promotion, that o is either a Foo instance or null, which enables us to eliminate ensure-guarded casts.

• if (e) S_t else S_f

The if-then-else statement naturally propagates the conditional information of $\mathcal{A}_t[\![e]\!]$ and $\mathcal{A}_f[\![e]\!]$ to the then- and else-statements, respectively. Furthermore we use the reachability information of these substatements to determine the after map of the if-statement. If the then-statement completes normally[2], denoted $\mathcal{C}[\![S_t]\!]$, and the else-statement *does not* complete normally then the after map of the if-statement is the after map of the then-statement. Likewise in the opposite case. If both the then- and the else-statements complete normally, the after map is simply the *join* of the substatements.

$$\begin{array}{rcl} \mathcal{B}[\![e]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{B}[\![S_t]\!] & = & \mathcal{A}_t[\![e]\!] \\ \mathcal{B}[\![S_f]\!] & = & \mathcal{A}_f[\![e]\!] \\ \mathcal{A}[\![\cdot]\!] & = & \begin{cases} \mathcal{A}[\![S_t]\!] & \text{if } \mathcal{C}[\![S_t]\!] \text{ and } \neg\mathcal{C}[\![S_f]\!] \\ \mathcal{A}[\![S_f]\!] & \text{if } \neg\mathcal{C}[\![S_t]\!] \text{ and } \mathcal{C}[\![S_f]\!] \\ \mathit{join}(\mathcal{A}[\![S_t]\!], \mathcal{A}[\![S_f]\!]) & \text{otherwise} \end{cases} \end{array}$$

• try S catch (T_1 v_1) S_1 ... catch (T_n v_n) S_n finally S_f

The transfer function for try-statements must soundly take into account the many possible paths through a try-statement. This is done by conservatively assuming that no new promotional information is obtained through the try and catch clauses and that the information for every updated target must be removed. The transfer function is thus

$$\begin{array}{rcl} \mathcal{B}[\![S]\!] & = & \mathcal{B}[\![\cdot]\!] \\ \mathcal{B}[\![S_i]\!] & = & \mathcal{B}[\![\cdot]\!] \setminus \{t \mapsto \bullet \mid t \in \mathsf{targets}(S)\} \\ \mathcal{B}[\![S_f]\!] & = & \mathcal{B}[\![\cdot]\!] \setminus \{t \mapsto \bullet \mid t \in \mathsf{targets}(S, S_1, \cdots, S_n)\}) \\ \mathcal{A}[\![\cdot]\!] & = & \mathcal{A}[\![S_f]\!] \end{array}$$

where targets computes the set of all updated targets of the given statements, which is simply the left-hand side targets in the statements.

5. INTEGRATION INTO JAVA

If we were to design a new language, we could use the promoted type as a primary source in type checking. Since we are extending an existing language we cannot do this without breaking compatibility. The linking of method and constructor invocations and field accesses is based on the static types of the receiver and arguments. If the static types change, the linked target might change, thus changing the semantics of existing code. For instance

```
void m(Foo foo) { ... }
void m(Bar bar) { ... }

Foo foo = ...
if (foo instanceof Bar)
  m(foo); // m(Bar) is linked
```

where the method m(Bar) is linked instead of m(Foo) by the use of type promotion. What we do instead is to use Guarded Type Promotion as a 'Plan B'. If the type checking of a construct fails using the declared types, we type check the construct again using the promoted types.

While this ensures backward compatibility, since only currently untypeable programs will use the promoted types, there is one issue that needs to be addressed. When the static types change because Guarded Type Promotion is needed, the linked target might change unexpectedly. Consider for instance the following code.

```
void m(Bar bar, Foo foo) { ... }
void m(Bar bar1, Bar bar2) { ... }

Foo foo = ...
if (foo instanceof Bar) {
  m(foo, foo);
}
```

Here promotion is needed for the first argument in order to type m(foo, foo) but as a side-effect the second argument is promoted as well, making the linked method m(Bar,Bar) and not m(Bar,Foo). Since this second promotion might be unintended, such a linking is considered ambiguous, requiring an explicit cast of one of the arguments. In our experiments only 2 invocations were considered ambiguous.

To make Guarded Type Promotion binary compatible [5] casts are automatically inserted for the inferred promotions in the generated .class files. If the casts were not inserted, the class-file verifier would need to include some sort of type promotion as well.

6. EXPERIMENTS

To test the effectiveness and precision of the Guarded Type Promotion rules we have implemented these in a Java 1.6 compiler[3]. As test cases we have used the OpenJDK6 project [7], 8 different Apache projects [4], and 11 various projects including our extended Java compiler and active projects from Google Code [2]. In total 5.2 MLOC containing >35000 casts.

By analysing the casts in these projects we have measured the frequency of the 4 cast groups described in Section 2 and the results are shown in Figure 2 and in the 'Cast Group' section of Table 1. As we can see, the percentage of guarded casts varies greatly between individual projects, between 0% and 65%, but on average guarded casts account for 24.3% of all casts.

To test the precision of our type promotion rules, our compiler has simulated the absence of casts to see how many of the guarded casts that would be inferred. The results are shown in the 'Precision' section of Table 1. Here we see that 94.7% of the guarded casts were promoted. The type promotion failed to infer the remaining casts for three

[2]A statement completes normally if it does not return or throw an exception.

[3]Available at `http://cs.au.dk/~jwbrics/java/`

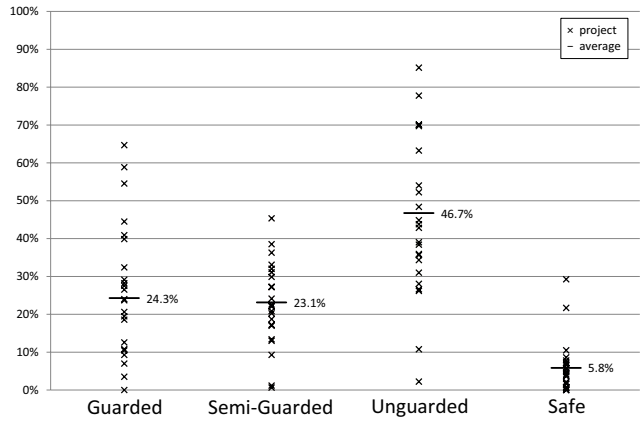

Figure 2: Cast Group Frequency

		%	#
Cast Group	Guarded	24.3	8624
	Semi-Guarded	23.1	8220
	Unguarded	46.7	16595
	Safe	5.8	2076
Precision	Promoted	94.7	8165
	Alias	2.4	211
	Dead	2.9	247
	Try	0.0	1
Expression Type	Promotable	87.8	7167
	Field Access	6.5	529
	Method Invocation	3.9	317
	Array Access	1.9	152

Table 1: Experimental Results

reasons. 1) The guarded information is hidden in an *alias* like in

```
boolean b = o instanceof Foo;
if (b) ...
```

2) The guard is hidden by a *dead* method, which is a method that does not complete normally, possibly given its arguments. For instance

```
assertTrue(o instanceof Foo);
... (Foo)o ...
```

where assertTrue does not complete normally given the argument false. 3) The guard is hidden by the inability to detect the needed guarding paths through a try-statement.

As shown in the 'Expression Type' section of Table 1, 87.8% of the guarded casts are directly promotable, that is, the guarded expressions are local variable or final field accesses. To be promotable the remaining expressions require a rewriting like

```
if (foo.getX() instanceof Bar) ...
```

into something like

```
X x = foo.getX();
if (x instanceof Bar) ...
```

7. RELATED & FUTURE WORK

Casts in Java have been targeted before by improvements of the type system. Most notably generics [10] which target a subset of the unguarded casts and make these safe. Also return type substitution introduced in Java 5 has the potential to eliminate unguarded casts for instance of calls to clone().

The IDEA [1] development environment has a feature based on a similar analysis which automatically inserts casts into the source code in some of the cases where expressions are guarded. This feature is purely pragmatic and casts both non-final field accesses and method invocations, though not array accesses. The feature does not take assignment into account and therefore also inserts unsound casts in some cases.

Similar work has also been done for dynamicly typed languages like Scheme [9, 8] and JavaScript [6] where conditional flow and runtime checks are taken into account in the typing rules.

In C# the as operator can be used to do a guarded promotion of a variable through this idiom:

```
var f = o as Foo;
if (f != null) {
  f.DoFooStuff();
}
```

This approach avoids the redundancy of naming Foo twice but instead requires the additional test for nullness.

Incorporating the data-flow analysis for Guarded Type Promotion into the type checker increases the space/time complexity of type checking. Future work lies in studying both the theoretical and practical cost of this added complexity. The soundness properties of Guarded Type Promotion, that the promoted type is always a subtype of the static type and a supertype of the dynamic type, are currently only conjectured. Future work lies in proving these properties formally. Furthermore further study is needed to determine to which extent the rules for Guarded Type Promotion and its integration into Java is comprehensible to programmers.

8. REFERENCES

[1] IntelliJ IDEA - The Most Intelligent Java IDE. JetBrains (2011), http://www.jetbrains.com/idea/

[2] Project Hosting on Google Code. Google (2011), http://code.google.com/hosting/

[3] von der Ahé, P.: Java SE 7 wish list (Dec 2006), http://blogs.sun.com/ahe/entry/java_se_7_wish_list

[4] The Apache Software Foundation. The Apache Software Foundation (2010), http://www.apache.org/

[5] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. Addison-Wesley Longman (2005)

[6] Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing Local Control and State Using Flow Analysis. In: ESOP. pp. 256–275 (2011)

[7] OpenJDK. Oracle Corporation (2010), http://openjdk.java.net/

[8] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: POPL. pp. 395–406 (2008)

[9] Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: ICFP. pp. 117–128 (2010)

[10] Torgersen, M., Hansen, C.P., Ernst, E.: Adding wildcards to the Java programming language. In: Journal of Object Technology. pp. 1289–1296. ACM Press (2004)

APPENDIX

A. EXAMPLES

In this section we present a few examples of how the data-flow analyses enables the typing of guarded casts. We show one example for each of the four kinds of guarded casts described in Section 2 as well as an example of an invalid cast.

A.1 if-Guarded Cast

In the first example we use an instanceof-expression to guard the use of the variable o:

```
1  int m(Object o) {
2    int x = 0;
3    if (o instanceof String) {
4      x = o.length();
5    }
6    return x;
7  }
```

Some of the computed type maps are:

$$\begin{aligned}
\mathcal{B}[\![2]\!] &= \{\} \\
\mathcal{A}_t[\![3]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{String}\}, \emptyset) \\
&\quad \}, \{\mathsf{String}\})\} \\
\mathcal{A}_f[\![3]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \{\mathit{typeHolder}(\mathsf{Object}, \emptyset, \{\mathsf{String}\}) \\
&\quad \}, \{\mathsf{String}\})\} \\
\mathcal{B}[\![4]\!] &= \mathcal{A}_t[\![3]\!] \\
\mathcal{A}[\![4]\!] &= \mathcal{B}[\![4]\!] \\
\mathcal{B}[\![6]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{String}\}, \emptyset), \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \emptyset, \{\mathsf{String}\}) \\
&\quad \}, \{\mathsf{String}\})\}
\end{aligned}$$

Initially, in $\mathcal{B}[\![2]\!]$, we have no promotional information. After the condition in line 3 we have two results. If true we have in $\mathcal{A}_t[\![3]\!]$ that o is known to be an instance of String and if false we have in $\mathcal{A}_f[\![3]\!]$ that o is known *not* to be an instance of String. In line 4 we have the same information as in $\mathcal{A}_t[\![3]\!]$ enabling the call of length() on o. After the if-statement, in line 6, we have the joined information of $\mathcal{A}[\![4]\!]$ and $\mathcal{A}_f[\![3]\!]$, that is, that o is either an instance of String or not. Note that since we have collected String as a declaration of interest this is *not* the same as no information. Such information could be used by later assignments like in the ensure-guarded cast example below.

A.2 Dead-if-Guarded Cast

The second example uses a getClass()-test together with reachability to guard the use of the variable o:

```
1  boolean m(Object o, Number n) {
2    if (o.getClass() != n.getClass()) {
3      return false;
4    }
5    return o.intValue() == n.intValue();
6  }
```

Some of the computed type maps are:

$$\begin{aligned}
\mathcal{B}[\![2]\!] &= \{\} \\
\mathcal{A}_t[\![2]\!] &= \mathcal{B}[\![2]\!] \\
\mathcal{A}_f[\![2]\!] &= \{ \\
&\quad \mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{Number}\}, \emptyset) \\
&\quad \}, \{\mathsf{Number}\}), \\
&\quad \mathcal{L}ocal(\mathsf{n}) \mapsto \mathit{targetInfo}(\mathsf{Number}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Number}, \{\mathsf{Object}\}, \emptyset) \\
&\quad \}, \{\mathsf{Object}\}) \\
&\quad \}
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![3]\!] &= \mathcal{A}_t[\![2]\!] \\
\mathcal{B}[\![5]\!] &= \mathcal{A}_f[\![2]\!]
\end{aligned}$$

Here we see that the guard in line 2 provides no information on true. On false, however, we extend the type map with the knowledge that o is an instance of Number and that n is an instance of Object, i.e. that n is not null. Since the body of the if-statement does not complete normally we have in $\mathcal{B}[\![5]\!]$ the same information as in $\mathcal{A}_f[\![2]\!]$ enabling the call to intValue() on o.

A.3 Ensure-Guarded Cast

In this example we ensure the type of o through both paths of an if-statement:

```
1  int m(Object o) {
2    if (!(o instanceof String)) {
3      o = o.toString();
4    }
5    return o.length();
6  }
```

Some of the computed type maps are:

$$\begin{aligned}
\mathcal{B}[\![2]\!] &= \{\} \\
\mathcal{A}_t[\![2]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \\
&\quad \{\mathit{typeHolder}(\mathsf{Object}, \emptyset, \{\mathsf{String}\})\}, \{\mathsf{String}\})\} \\
\mathcal{A}_f[\![2]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \\
&\quad \{\mathit{typeHolder}(\mathsf{Object}, \{\mathsf{String}\}, \emptyset)\}, \{\mathsf{String}\})\} \\
\mathcal{B}[\![3]\!] &= \mathcal{A}_t[\![2]\!] \\
\mathcal{A}[\![3]\!] &= \{\mathcal{L}ocal(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{String}\}, \emptyset), \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \emptyset, \{\mathsf{Object}\}) \\
&\quad \}, \{\mathsf{String}\})\} \\
\mathcal{B}[\![5]\!] &= \mathit{join}(\mathcal{A}_f[\![2]\!], \mathcal{A}[\![3]\!]) = \mathcal{A}[\![3]\!]
\end{aligned}$$

After false of the condition in line 2 we know o to be an instance of String. After true we know that o is *not* an instance of String but that String *is* a declaration of interest. This is used in the assignment in line 3 which collects the information in $\mathcal{A}[\![3]\!]$ that o is either an instance of String or null. The information in $\mathcal{B}[\![5]\!]$ is the join of $\mathcal{A}_t[\![2]\!]$ and $\mathcal{A}[\![3]\!]$ which is same as $\mathcal{A}[\![3]\!]$ itself, i.e. that o is a subtype of String (possibly null) enabling the call to length() on o.

A.4 while-Guarded Cast

In this example we guard the use of c through a loop over the parent relation of the component. Code patterns like this occur several times in the AWT/Swing framework.

```
1  Window m(Component c) {
2    while (c != null
3      && !(c instanceof Window))
4    {
5      c = c.getParent();
6    }
7    return c;
8  }
```

Some of the computed type maps are:

$$\begin{aligned}
\mathcal{B}[\![2]\!] &= \{\} \\
\mathcal{A}_t[\![2]\!] &= \{\mathcal{L}ocal(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Component}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \{\mathsf{Component}\}, \emptyset) \\
&\quad \}, \{\mathsf{Component}\})\} \\
\mathcal{A}_f[\![2]\!] &= \{\mathcal{L}ocal(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Component}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \emptyset, \{\mathsf{Component}\}) \\
&\quad \}, \{\mathsf{Component}\})\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{B}[\![3]\!] &= \mathcal{A}_t[\![2]\!] \\
\mathcal{A}_t[\![3]\!] &= \{\mathit{Local}(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Component}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \{\mathsf{Component}\}, \{\mathsf{Window}\}) \\
&\quad \}, \{\mathsf{Component}, \mathsf{Window}\})\} \\
\mathcal{A}_f[\![3]\!] &= \{\mathit{Local}(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Component}, \{ \\
&\quad \{\mathit{typeHolder}(\mathsf{Component}, \{\mathsf{Component}, \mathsf{Window}\}, \emptyset) \\
&\quad \}, \{\mathsf{Component}, \mathsf{Window}\})\} \\
\mathcal{B}[\![5]\!] &= \mathcal{A}_t[\![3]\!] \\
\mathcal{A}[\![5]\!] &= \{\mathit{Local}(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \emptyset, \{\mathsf{Component}\}), \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \{\mathsf{Component}\}, \emptyset) \\
&\quad \}, \{\mathsf{Component}, \mathsf{Window}\})\} \\
\mathcal{B}[\![7]\!] &= \{\mathit{Local}(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \{\mathsf{Window}\}, \emptyset), \\
&\quad \mathit{typeHolder}(\mathsf{Component}, \emptyset, \{\mathsf{Window}\}) \\
&\quad \}, \{\mathsf{Component}, \mathsf{Window}\})\}
\end{aligned}
$$

The loop condition falls in two parts. The first part in line 2 determines nullness of c and the second part determines whether c is a Window instance. In line 5 we thus know from $\mathcal{A}_t[\![3]\!]$ that c *is* an instance of Component but not of Window. After the loop in line 7 we know the join of $\mathcal{A}_f[\![2]\!]$ and $\mathcal{A}_f[\![3]\!]$, i.e. that c is either null or an instance of Window. If we had not collected the nullness information in line 2 we would not have known c to be a subtype of Window in line 7 since the join of $\mathcal{A}_f[\![2]\!]$ and $\mathcal{A}_f[\![3]\!]$ then would have been empty.

A.5 Invalid Cast

The last example shows how invalid casts can be detected through a sequence of contradicting guards.

```
1  void m(Object o) {
2    o = (String)o;
3    if (o instanceof Integer) {
4      if (o instanceof String) {
5        ...
6      }
7    }
8  }
```

Some of the computed type maps are:

$$
\begin{aligned}
\mathcal{B}[\![2]\!] &= \{\} \\
\mathcal{B}[\![3]\!] &= \mathcal{B}[\![2]\!] \\
\mathcal{A}_t[\![3]\!] &= \{\mathit{Local}(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{Integer}\}, \emptyset) \\
&\quad \}, \{\mathsf{Integer}\})\} \\
\mathcal{A}_f[\![3]\!] &= \{\mathit{Local}(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \emptyset, \{\mathsf{Integer}\}) \\
&\quad \}, \{\mathsf{Object}\})\} \\
\mathcal{B}[\![4]\!] &= \mathcal{A}_t[\![3]\!] \\
\mathcal{A}_t[\![4]\!] &= \{\mathit{Local}(\mathsf{o}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{Integer}, \mathsf{String}\}, \emptyset) \\
&\quad \}, \{\mathsf{Integer}, \mathsf{String}\})\} \\
\mathcal{A}_f[\![4]\!] &= \{\mathit{Local}(\mathsf{c}) \mapsto \mathit{targetInfo}(\mathsf{Object}, \{ \\
&\quad \mathit{typeHolder}(\mathsf{Object}, \{\mathsf{Integer}\}, \{\mathsf{String}\}), \\
&\quad \}, \{\mathsf{Integer}, \mathsf{String}\})\}
\end{aligned}
$$

Here we see that line 2 is *not* considered a guard, providing no promoted information about o in $\mathcal{B}[\![3]\!]$. The true branch of line 3 provides the information that o is an instance of Integer and the true branch of line 4 expands this by adding String to the set of true declarations. Now since all type holders of $\mathcal{A}_t[\![4]\!]$ are *invalid* — the sole type holder states that o is an Integer and a String instance at the same time — an 'unreachable code' error is given for line 4.